Pyodide Components

Pyodide Components

Dec 04, 2022

CONTENTS

1 New directory 3

2 First files 5

3 Recap 7

4 Python Setup 9

5 NodeJS Setup 13

6 Vite Cleanup 17

7 Re-arrange favicon 19

8 Setup Vitest 21

9 Setup `pytest` 25

10 Playwright Tests 27

11 Downloader 33

12 Import Pyodide 37

13 Faster Pyodide Testing 41

14 Simple Component Registry 43

15 Mocking `fetch` 45

16 Worker Script 49

17 Worker Messaging 53

18 Worker to Main 55

19 Playwright Pyodide 61

20 Refactor Speedup 63

21 Registry Revisited 65

22 Simple Components 67

23 Loadable Apps 75

24 Component instances 79

25 TODO 85

- "Joyful"
  - Use modern tooling to "fail faster" and "stay in the flow"
    - IDE
    - Linter/formatter
    - But no type checker, for the target audience of this series, a step too far
  - You can do bare-bones tooling as well, we'll show that mode

Let's do a small first step, just to get things in place. We'll make a directory, under version control, with a README and a `.gitignore` file.

CHAPTER ONE

NEW DIRECTORY

This series is focused on teaching a “joyful” development style for Pyodide. It’s secondary purpose is pitching an idea about a framework for Python-based custom elements as “components.”

So let’s jump right into that and setup `pyodide-components` as our project workspace:

```
$ mkdir pyodide-components
$ cd pyodide-components
```

We also want to initialize this as a Git repo:

```
$ git init
```

CHAPTER TWO

FIRST FILES

First, add a simple README.md file. At a minimum, it will help the future you remember what this directory was about.

```
# Pyodide Components

Learn a "joyful" way of Pyodide development while writing a simple framework
for Python custom elements.
```

We'll also add a .gitignore file. Later we'll add entries to it:

```
$ touch .gitignore
```

Finish by adding these to the repo and committing:

```
$ git add README.md .gitignore
$ git commit -m"Start project"
```

CHAPTER

THREE

RECAP

Yep, this was some pro forma prep work, from the command-line. We could have used our IDE, but we're going to stay old-school for the first few steps. No "joyful" yet. But a step forward before setting up our Python and NodeJS workspaces.

CHAPTER

FOUR

PYTHON SETUP

We want a nice home for our Python project: metadata, dependencies, and sharing. We'll choose the really-modern Python approach:

- `pyproject.toml`
- `setuptools` as the build backend (no Poetry/Hatch, just regular pip)
- An editable install

4.1 Why?

Python certainly has. . . lots of options. Why this as-yet-little-known approach?

Why an "editable install"? The Python testing community encourages an `src` layout of your project, as recommended by others. When you make your workspace into a proper "package", you can import from anywhere. . . once the package is an "editable install".

Why not just a `requirements.txt`? This won't get you into "proper package" mode. Ditto for `pipenv` which is more about applications than packages.

Why not `setup.py`? The Python world is (hopefully) moving away from that, to a world with `pyproject.toml` as the central configuration spot.

Why not Poetry or Hatch? This series is trying to stay on a mainstream path. `pip` is still the king of the hill. Now that `setuptools` is a valid `pyproject.toml` backend, that's a good happy path for beginners. Also, `setuptools` directly supports the `src` layout described above.

Dear heavens, I hope one day to never need to explain that again.

4.2 Python editable install with `pyproject.toml`

We're doing a "joyful Python" project. That means coding through the lens of a *test*. `pytest` best practices say to make a package and then do an "editable install."

We'll follow the `setuptools` page above, starting with an empty `pyproject.toml`.

```
$ touch pyproject.toml
```

4.2.1 Build backend

In the first section of the TOML file, we need to tell our packaging tool what build backend to use. There are lots of packaging tools – we're using `pip`. There are a number of backends – we're using `setuptools`:

```
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
```

4.2.2 Project metadata

Next, we'll tell our tooling – and the world – a little about our project. Add this section to the `pyproject.toml` file:

```
[project]
name = "pyodide_components"
version = "0.0.1"
requires-python = ">=3.10"
license = {text = "BSD 3-Clause License"}
classifiers = [
    "Programming Language :: Python :: 3",
]
dependencies = [
    "sphinx",
]
```

We're doing the minimum for dependencies for now: just Sphinx, as a way to ensure our installation works.

4.3 Project directory

Let's put some empty code into our project directory. We said we were adopting the src layout. Make an empty package file as a starter:

```
$ touch src/pyodide_components/__init__.py```
```

4.4 Virtual environment

We want to follow best practices and work in a virtual environment. Make one in the project folder, then upgrade the `pip` and `setuptools` it uses:

```
$ python -m venv .venv
$ .venv/bin/pip install --upgrade pip setuptools
```

4.5 Editable install

We have a virtual environment. We have a `pyproject.toml` that defines our package. But the virtual environment needs to know about our package.

Let's do an editable install. This put `pyodide_components` in the virtual environment's `site-packages`. But, as basically a symbolic link back to the `src/pyodide_components` directory:

```
$ .venv/bin/pip install -e .
```

If this worked correctly, you now have a `src/pyodide_components.egg-info` directory. You also have `sphinx-quickstart` in your virtual environment's `bin`:

```
$ ls src/pyodide_components.egg-info
PKG-INFO              requires.txt
SOURCES.txt             top_level.txt
dependency_links.txt
$ ls .venv/bin/sphinx*
.venv/bin/sphinx-apidoc
.venv/bin/sphinx-autogen
.venv/bin/sphinx-build
.venv/bin/sphinx-quickstart
```

To confirm that we can import `pyodide_components` outside its source directory:

```
$ .venv/bin/python -c "import pyodide_components"
```

4.6 Cleanup

There we go, a modern Python workspace. Let's clean up a bit.

First, add some exclusions to your `.gitignore` file:

```
*.egg-info/
.venv
__pycache__/
```

Now commit your work:

```
$ git add .gitignore pyproject.toml src
$ git commit -m"Python project workspace setup"
```

CHAPTER

FIVE

NODEJS SETUP

We also want a nice home for the JavaScript side. Nothing too fancy. But also nothing too austere.

In this section we'll get a NodeJS project setup with a minimum based around the Vite tooling.

:::{note} NodeJS 18.3.0 or higher This tutorial presumes you are using NodeJS with the LTS (at time of writing) or higher. Why? We need `fetch`. Otherwise, install `node-fetch` yourself in an older NodeJS. :::

5.1 What? What?

"NodeJS? Vite? WTH? I'm here for Python, not crazy JS frontend lolz."

Yes, good point. BUT…the world of frontends has gotten very interesting and productive. We're doing a "vanilla" project: no TypeScript and no frameworks.

But even so, we can benefit from modern tooling. This is the "joyful" path.

We want "joyful Python" *and* "joyful JavaScript". For the JavaScript *development*, *joyful* primarily means using NodeJS in an IDE instead of flipping to the browser all the time. There's lots of great tooling in modern frontends. Let's use some of it without going crazy.

We will center our strategy on Vite and its test runner named Vitest. Vite is super-fast tooling for JavaScript applications, giving a fantastic developer experience, even for plain-old-JS projects.

5.2 NodeJS setup

This series presumes you have a NodeJS installation. Let's confirm this:

```
$ node -v
```

5.3 Setup a Vite project

The NodeJS equivalent of `pyproject.toml` is the `package.json` file. We'll create that by asking a Vite scaffold to make us a "hello world" project for vanilla JS.

Start with the `npm` command, which is like `pip` but for NodeJS. It has a mode like `pipx` where it can execute something that isn't locally installed:

```
$ npm create vite@latest pyodide-components -- --template vanilla
```

It generated files into a subdirectory. Let's copy all of that up into our directory (while preserving our `.gitignore`), then remove that scaffold directory:

```
$ cp pyodide-components/.gitignore >> .gitignore
$ rm pyodide-components/.gitignore
$ cp -r pyodide-component/* .
$ rm -rf pyodide-components
```

5.4 Vite cleanup

Sorry, we're going to have to talk about some things. Just a few, not too scary.

The scaffold generated a sample application: `index.html`, `counter.js`, `javascript.svg`, `style.css`, and `main.js`. It also generated `package.json` and a directory `public` for absolute-referenced static paths.

We'll later remove much of that. For now, let's re-arrange some things to fit our project structure:

- Move `index.html`, `counter.js`, `javascript.svg`, `style.css`, and `main.js`…
- …to `src/pyodide_components`

Then, edit the `package.json` to reflect this directory structure:

```
  "scripts": {
    "dev": "vite serve src/pyodide_components/",
    "build": "vite build --outDir=../../dist src/pyodide_components/",
    "preview": "vite preview"
  },
```

5.5 Install and run

We're now in-place. Let's install our dependencies:

```
$ npm install
```

This creates a `node_modules` directory with your one dependency – vite – and its dependencies (around 16.) 20 megabytes-ish. Not terrible…after all, the virtual environment's `site-packages` with Sphinx is 90 Mb.

You can now run the dev server:

```
$ npm run dev
```

This launches a live-reload server running at a URL. Click on the URL and it launches in a browser. Make a change in the HTML or JS and you'll see the browser is updated. It's fast!

Or, generate a build in a top-level `dist` directory:

```
$ npm run build
```

This creates a shippable site at `dist`.

5.6 Add to git

We’ve added some files, obviously, so let’s do a checkpoint:

```
$ git add .gitignore pyproject.toml package* src public
$ git commit -m"NodeJS project workspace setup"
```

5.7 Wrapup

Good first step, from the command-line. Let’s switch over to our IDE, install some dependencies, and get some workflows going.

CHAPTER

SIX

VITE CLEANUP

The scaffold gave us a demo application. But lots of that stuff will get in the way later. Let's do a quick cleanup, while ensuring that the demo still works in both modes:

- Dev mode with live reloading
- Build mode combined with the preview server

CHAPTER

SEVEN

RE-ARRANGE FAVICON

The scaffold puts a `vite.svg` favicon in the `public` folder as a demo of a build feature. Let's reduce that complexity.

First, make a `src/pyodide_components/static` directory. Then, move `public/vite.svg` to `src/pyodide_components/static/vite.svg`:

```
$ mkdir src/pyodide_components/static
$ mv public/vite.svg src/pyodide_components/static/vite.svg
$ rmdir public
```

Then, change `index.html` just slightly to stop using an absolute URL:

```
<link rel="icon" type="image/svg+xml" href="./static/vite.svg" />
```

7.1 Strip down HTML and static

We'll remove some stuff from both the HTML and the JS. First, `index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <link rel="icon" type="image/svg+xml" href="./static/vite.svg"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>Pyodide Components</title>
</head>
<body>
<h1>Pyodide Components</h1>
<script type="module" src="main.js"></script>
</body>
</html>
```

:::{note} Easy formatting with Prettier Life is too short to manually format code. Do `npm install prettier` and let your smart editor do formatting. :::

Now a vastly-stripped-down `main.js`:

```
export const PYODIDE_COMPONENTS = "Hello";
```

This means you can delete `style.css`, `counter.js` and `javascript.svg`.

7.2 Dev server

Go back to `package.json` and run your `dev` script using your IDE, or from the command-line:

```
$ npm run dev
```

You'll now see a very stripped down page. Only the favicon `vite.svg` and `main.js` files are loaded. Of course, as you change the `index.html` (or anything it loads), the browser is updated.

7.3 Build and preview

Does the bundler still work? Run the `build` script and take a look at the `dist` folder. You can then run the `preview` script and click on the URL. You'll see a browser view of the statically-generated contents.

CHAPTER

EIGHT

SETUP VITEST

Start doing basic JavaScript testing using Vitest.

8.1 Why Testing?

I'm a big fan of "test-first" development. Not TDD – that kind of implies an "eat your vegetables" approach where the goal is quality.

Instead, I do my thinking and working inside a test because it is more *convenient*. Switching to a browser, hitting reload, `console.log()` everywhere, struggling with a debugger. This does not spark joy. Instead, let tooling provide a better development experience.

For JavaScript, this means:

- Live in NodeJS, not the browser
- Writing code in exported chunks that can easily be imported in a test
- Use the test as a kind of REPL
- Run the tests under the debugger, to easily stop in a context and poke around

8.2 Why Vitest?

I don't want to make folks adopt some heavy JS tooling hellscape. Vitest is nice because it is fast, lighter-weight, modern (ESM-first), and has good tooling support.

To be clear: we're not using a JavaScript framework such as Vue. We're using Vite and Vitest with vanilla JS.

8.3 Launch your IDE

At this point we'll move from the command line to "your editor of choice." I'll refer to it as "the IDE". In my case, this means PyCharm Professional but smart editors and IDEs have all become quite good at "modern tooling."

8.4 Installing Vitest

Let's install Vitest and its companion `@vitest/ui` package that puts a pretty UI on testing.

If you're familiar with NodeJS development, you'll know: `npm` is used for installing a package and recording the dependency in `package.json`. Thus:

```
$ npm install vitest @vitest/ui
```

Why not `npm install -D` to record this as a development dependency? This is a tutorial series, not really a releasable-project. We'll simplify by only having one set of dependencies.

8.5 Hello World JS

We'll start with a new file named `src/pyodide_components/main.mjs`. That already brings up a question – what's the `.mjs` extension? This helps flag to various tooling (NodeJS, web servers, etc.) that this file uses ECMAScript Modules (ESM) syntax. With this, no module bundlers are needed when serving to a browser, as ESM Imports are now well-supported.

Here's an in-depth ESM guide to ESM for library authors.

The file is pretty empty:

```
export const PYODIDE_COMPONENTS = "Hello";
```

Its purpose is only to get us started testing.

However, it already has a second head-scratcher: why no "document.addEventListener" for `DOMContentLoaded`? As it turns out, in browsers, you can load with `<script defer>` and get the same effect. This is recommended in Jake's HTTP 203 episode.

8.6 Hello World Test

We have exported the `const`. Let's write a simple test in `tests/main.test.js` which imports and checks it:

```
import {expect, test} from "vitest";
import {PYODIDE_COMPONENTS} from "../src/pyodide_components/main.mjs";

test("Hello", () => {
    expect(PYODIDE_COMPONENTS).to.equal("Hello");
});
```

The filename `main.test.js` uses the `.test.` convention. These files are usually alongside the source, but in Python the convention is different. Also, we don't want those files shipping in the wheel we might build.

Why no `.mjs` extension? This file will only be executed on the NodeJS side.

The test does an ESM-style import of the exported consistent. It then has a single test, using Vitest's bundled Chai assertion library.

8.7 Running Vitest

Let's edit the `scripts` block in `package.json` and add an entry to run `vitest`:

```
    "test": "vitest --ui"
```

You can run Vitest against this:

```
$ npm test
```

Notice that this isn't `npm run test`? `npm` has some built-in shortcuts that it treats as first-class operations, such as `test`.

This goes into watch mode and brings up a nice UI in a browser. Changes trigger a test run which updates the browser.

As a note, Vitest is FAST. It also has a watch mode to be really fast.

8.8 Wrapup

As usually, commit all the changes.

If you are using an IDE (VS Code, PyCharm) then you have good Vitest integration. This replaces the need for running the `npm` script, the dev server, and looking in a browser.

We'll use an IDE for the rest of this series.

CHAPTER

NINE

SETUP PYTEST

We also want testing for our Python code, so let's get `pytest` installed.

9.1 Install and Config

To start, we'll add `pytest` to the dependencies in `pyproject.toml`. Really, they should go in dev dependencies, but this project really just a tutorial. Let's go ahead and add all the dependencies we'll need for the next few steps:

```
dependencies = [
    "sphinx",
    "myst-parser",
    "furo",
    "pytest",
    "playwright",
    "html5lib",
    "pytest-playwright",
    "starlette",
    "uvicorn",
    "watchfiles",
    "anyio",
    "httpx"
]
```

After adding, we do:

```
$ pip install -e .
```

We'll also start the process of centralizing our pytest options. Instead of `pytest.ini`, we'll use `[tool.pytest.ini_options]` in `pyproject.toml`. As an example, add this to `pyproject.toml` to configure strict marker usage in `pytest`:

```
[tool.pytest.ini_options]
addopts = "--strict-markers"
```

9.2 First Python Code and Test

As before, we'll put a little tracer in our Python code and write a simple test.

In `src/pyodide_components/__init__.py`:

```
PYODIDE_COMPONENTS = "Hello"
```

Then, in `tests/test_init.py`:

```
from pyodide_components import PYODIDE_COMPONENTS


def test_hello():
    assert PYODIDE_COMPONENTS == "Hello"
```

Running pytest shows that the test passes.

CHAPTER

TEN

PLAYWRIGHT TESTS

Add end-to-end (E2E) testing in a real browser.

10.1 Why? What?

We have an HTML page. Most of what we want to do can be done with a fake DOM. But we also want to test in a real browser, particularly when Pyodide integration lands.

We will especially want to confirm that our `build` step produces a bundle that works. It's a place that can break a lot, particularly across browsers.

We'll use Playwright for this. In particular, we'll extend our `pytest` testing to also use `pytest-playwright`.

10.2 Setup `pytest-playwright`

First, make sure `pytest` and `pytest-playwright` are in `pyproject.toml`. We added this in one of the first steps.

Next, run `playwright install` from the command line. `playwright` is a "command" that was installed into your `.venv/bin`. This gets browser binaries on your local system.

We're going to make some pytest fixtures to speed up our testing. Add a file `src/pyodide_components/fixtures.py` – empty for now. Then, add `tests/conftest.py` to load them, with:

```
pytest_plugins = "pyodide_components.fixtures"
```

10.3 Fixtures and tests

We want to run Playwright. But we don't want to have to fire up an HTTP server during our tests, just to serve files from disk. Instead, we'll use Playwright's "network interceptor" approach to catch HTTP requests and handle them from our files. And we'll do that in a fixture that installs it.

Which means – you guessed it – a *test* for the fixture we're going to write. The `tests/test_fixtures.py` shows this in action:

```
"""Ensure the test fixtures work as expected."""
from typing import cast

from playwright.sync_api import Page
from playwright.sync_api import Route
```

(continues on next page)

(continued from previous page)

```python
from pyodide_components import STATIC
from pyodide_components.fixtures import DummyPage
from pyodide_components.fixtures import DummyRequest
from pyodide_components.fixtures import DummyResponse
from pyodide_components.fixtures import DummyRoute
from pyodide_components.fixtures import route_handler


def test_dummy_request() -> None:
    """Ensure the fake Playwright request class works."""
    dummy_request = DummyRequest(url="/dummy")
    result = dummy_request.fetch(dummy_request)
    assert result.dummy_text == "URL Returned Text"


def test_dummy_response() -> None:
    """Ensure the fake Playwright response class works."""
    dummy_response = DummyResponse(dummy_text="test dummy response")
    assert dummy_response.text() == "test dummy response"
    assert dummy_response.body() == b"test dummy response"
    assert dummy_response.headers["Content-Type"] == "text/html"


def test_dummy_route() -> None:
    """Ensure the fake Playwright route class works."""
    dummy_request = DummyRequest(url="/dummy")
    dummy_route = DummyRoute(request=dummy_request)
    dummy_route.fulfill(
        body=b"dummy body", headers={"Content-Type": "text/html"}, status=200
    )
    assert dummy_route.body == b"dummy body"
    assert dummy_route.headers["Content-Type"] == "text/html"  # type: ignore


def test_route_handler_fake_good_path() -> None:
    """Fake points at good path in ``examples``."""
    # We are testing the interceptor, because the hostname is "fake".
    dummy_request = DummyRequest(url="https://fake/static/vite.svg")
    dummy_page = DummyPage(request=dummy_request)
    dummy_route = DummyRoute(request=dummy_request)
    route_handler(
        cast(Page, dummy_page),
        cast(Route, dummy_route),
    )
    if dummy_route.body:
        assert dummy_route.status == "200"
        with open(STATIC / "vite.svg", "rb") as f:
            body = f.read()
            assert dummy_route.body == body
```

(continues on next page)

(continued from previous page)

```python
def test_route_handler_non_fake() -> None:
    """Not fake thus not interceptor, but simulating network request."""
    dummy_request = DummyRequest(url="https://good/static/vite.svg")
    dummy_page = DummyPage(request=dummy_request)
    dummy_route = DummyRoute(request=dummy_request)
    route_handler(
        cast(Page, dummy_page),
        cast(Route, dummy_route),
    )
    assert dummy_route.body == b"URL Returned Text"


def test_route_handler_fake_bad_path() -> None:
    """Fake points at bad path in ``examples``."""
    dummy_request = DummyRequest(url="https://fake/staticxx")
    dummy_page = DummyPage(request=dummy_request)
    dummy_route = DummyRoute(request=dummy_request)
    route_handler(
        cast(Page, dummy_page),
        cast(Route, dummy_route),
    )
    assert dummy_route.status == "404"
```

We then implement the `src/pyodide_components/fixtures.py` file:

```python
"""Automate some testing."""
from __future__ import annotations

from dataclasses import dataclass
from dataclasses import field
from mimetypes import guess_type
from urllib.parse import urlparse

import pytest
from playwright.sync_api import Page
from playwright.sync_api import Route

from pyodide_components import HERE


@dataclass
class DummyResponse:
    """Fake the Playwright ``Response`` class."""

    dummy_text: str = ""
    headers: dict[str, object] = field(
        default_factory=lambda: {"Content-Type": "text/html"}
    )
    status: int | None = None

    def text(self) -> str:
        """Fake the text method."""
```

(continues on next page)

(continued from previous page)

```
        return self.dummy_text

    def body(self) -> bytes:
        """Fake the text method."""
        return bytes(self.dummy_text, "utf-8")


@dataclass
class DummyRequest:
    """Fake the Playwright ``Request`` class."""

    url: str

    @staticmethod
    def fetch(request: DummyRequest) -> DummyResponse:
        """Fake the fetch method."""
        return DummyResponse(dummy_text="URL Returned Text")


@dataclass
class DummyRoute:
    """Fake the Playwright ``Route`` class."""

    request: DummyRequest
    body: bytes | None = None
    status: str | None = None
    headers: dict[str, object] | None = None

    def fulfill(self, body: bytes, headers: dict[str, object], status: int) -> None:
        """Stub the Playwright ``route.fulfill`` method."""
        self.body = body
        self.headers = headers
        self.status = str(status)


@dataclass
class DummyPage:
    """Fake the Playwright ``Page`` class."""

    request: DummyRequest


def route_handler(page: Page, route: Route) -> None:
    """Called from the interceptor to get the data off disk."""
    this_url = urlparse(route.request.url)
    this_path = this_url.path[1:]
    is_fake = this_url.hostname == "fake"
    headers = dict()
    if is_fake:
        # We should read something from the filesystem
        this_fs_path = HERE / this_path
        if this_fs_path.exists():
```

(continues on next page)

(continued from previous page)

```
            status = 200
            mime_type = guess_type(this_fs_path)[0]
            if mime_type:
                headers = {"Content-Type": mime_type}
            body = this_fs_path.read_bytes()
        else:
            status = 404
            body = b""
    else:
        # This is to a non-fake server. Only for cases where the
        # local HTML asked for something out in the big wide world.
        response = page.request.fetch(route.request)
        status = response.status
        body = response.body()
        headers = response.headers

    route.fulfill(body=body, headers=headers, status=status)

@pytest.fixture
def fake_page(page: Page) -> Page:  # pragma: no cover
    """On the fake server, intercept and return from fs."""

    def _route_handler(route: Route) -> None:
        """Instead of doing this inline, call to a helper for easier testing."""
        route_handler(page, route)

    # Use Playwright's route method to intercept any URLs pointed at the
    # fake server and run through the interceptor instead.
    page.route("**", _route_handler)

    # Don't spend 30 seconds on timeout
    page.set_default_timeout(5000)
    return page
```

10.4 `index.html` and test

We want a home page with HTML loads our JS and updates a DOM. We'll start with a test at `tests/test_pages.py` and a first test for this index page.

Our fixture's interceptor catches anything to `http://fake/` and maps the rest of the path to the filesystem, rooted at `src/pyodide_components`. So we'll put an `index.html` there. Our test starts like this:

```
from playwright.sync_api import Page


def test_index(fake_page: Page):
    """Use Playwright to do a test on Hello World."""
    # Use `PWDEBUG=1` to run "head-ful" in Playwright test app
    url = "http://fake/index.html"
```

(continues on next page)

(continued from previous page)

```
    fake_page.goto(url)
    assert fake_page.title() == "Pyodide Components"
```

Not bad, and the test runs reasonably fast – for now . We also have good debugger support.

CHAPTER

ELEVEN

DOWNLOADER

We don't want to download Pyodide every time we run a test or open a page. Let's write a little downloader script and register it as a console app.

11.1 The Theory

We want Pyodide locally. It's cumbersome to download and extract, so we'll automate it with a script.

We'll store this in `src/pyodide_components/pyodide` with an entry in `.gitignore` to ensure it doesn't get checked in. Why under `src`? We'll explain in a bit.

11.2 The Code

We add the code to `src/pyodide_components/downloader.py`:

```
"""Downloads Pyodide and extracts to correct place."""
from pyodide_components import HERE

"""Automation scripts for getting setup."""
import os
import tarfile
from pathlib import Path
from shutil import copytree, rmtree
from tempfile import TemporaryDirectory

from urllib3 import PoolManager


def get_pyodide():
    print("Getting Pyodide")
    base_url = "https://github.com/pyodide/pyodide/releases/download"
    url = f"{base_url}/0.22.0a1/pyodide-0.22.0a1.tar.bz2"
    http = PoolManager()
    r = http.request('GET', url)
    with TemporaryDirectory() as tmp_dir_name:
        os.chdir(tmp_dir_name)
        tmp_dir = Path(tmp_dir_name)
        temp_file = tmp_dir / "pyodide.tar.bz2"
        temp_file.write_bytes(r.data)
```

(continues on next page)

(continued from previous page)

```
        tar = tarfile.open(temp_file)
        tar.extractall()
        target = HERE / "pyodide"
        if target.exists():
            rmtree(target)
        copytree(tmp_dir / "pyodide", target)


if __name__ == '__main__':
    get_pyodide()
```

That code depends on HERE. Let's add it to `src/pyodide_components/__init__.py`:

```
from pathlib import Path

HERE = Path(__file__).parent
```

11.3 Running it

It's a function that is run from a `__main__` block when you execute this in a virtual environment:

```
$ python -m pyodide_components.downloader
```

When run, it creates a directory at `src/pyodide_components/pyodide`. We should also add that directory to our `.gitignore`:

```
src/pyodide_components/pyodide
```

11.4 Add `node-fetch` dependency

If you fire up the `dev` server in `package.json`, you'll see a warning:

```
The following dependencies are imported but could not be resolved:

  node-fetch (imported by /somepath/pyodide-components/src/pyodide_components/pyodide/
↪pyodide.mjs)
```

Vite is doing some static analysis and notices that Pyodide's JS depends on a package called `node-fetch`. We can silence this by installing it:

```
$ npm install -S node-fetch
```

11.5 Places for improvement

We won’t spend too much time “hardening” this, as it is a just a means to an end. We could though:

- Add these instructions to a README
- Write a test with some mocks to prove the logic works
- Not hard-coding the URL path and version

CHAPTER

TWELVE

IMPORT PYODIDE

Let’s write some code that imports and runs Pyodide, then write a Vitest test.

12.1 Why?

It’s not a lot of fun doing Pyodide HTML/JS development the “normal” way. You write some code, reload your browser, open the console, and look for `console.log`. If you’ve learned how to use the browser’s debugger, that can help.

IDEs are good at running and debugging code. Since Vitest is a NodeJS application, you can get a much nicer development experience. Write a test, put a breakpoint in your Pyodide JS code, and stop right there. No browser involved.

12.2 Test Code

Let’s make a `tests/worker.test.js` that loads an `initialize` function and runs it:

```
import {expect, test} from "vitest";
import {initialize} from "../src/pyodide_components/worker.mjs";

test("Load and initialize Pyodide", async () => {
    const pyodide = await initialize();
    expect(pyodide).to.equal(2);
});
```

:::{note} `async` and `await`

Note the use of `async` on the arrow function and `await` when calling our function. We’ll use the async flavors of Pyodide wherever possible. :::

When you run this test, it fails. Good! Let’s go write the implementation.

12.3 Worker Code

For now the worker in `src/pyodide_components/worker.js` is really simple. One function which loads Pyodide and runs a Python expression:

```
import {loadPyodide} from "./pyodide/pyodide.mjs";

export async function initialize() {
    const pyodide = await loadPyodide();
    return pyodide.runPythonAsync("1+1");
}
```

:::{note} `.mjs` extension Pyodide puts its ESM-compatible files with a `.mjs` file extension. :::

When you run the test, you'll see it takes about 2 seconds to execute. But it runs! Even better, it runs in NodeJS, no need to leave your tooling to go to another application (the browser.)

12.4 Watch mode

Vitest has a watch mode which acts as a killer feature. When you ran the tests from the command-line, you saw it went into server mode, with a web UI for test output.

But the server mode is also watching for changes on the filesystem. If you change your code, or your tests, it automatically re-runs. Not just that, it re-runs without stopping the NodeJS process. Instead, it uses "hot module replacement" (HMR) to patch the running process with the changes.

But there's even more. Vitest does some analysis to determine which tests need to re-run, based on what changed. It's a very fast, "joyful" way to develop.

This mode is available in the IDE as well, which makes it even more joyful. Tracebacks have links to the line with the problem. The experience has an even more joyful mode.

12.5 Debugging and watch mode

We're developing using:

- NodeJS instead of the browser.
- Smart tooling such as an IDE.
- From a small test file, where we can poke around.
- A watch mode that's very smart about what changed.

Let's run it all under the NodeJS debugger. Vitest is so fast, you won't notice the speed hit. The nice part: if something needs investigating, you just set a breakpoint. When the test re-runs, it stops on that line and you can poke around.

12.6 Future work: speedups

All of this waxing-poetic about super-fast Vitest testing is obscured by a sad fact. Pyodide initialization is so slow, it eats up all those gains. Especially when you have multiple tests. Each test has to do the Pyodide initialization.

We'll tackle this in later steps. It's key to a productive Pyodide development experience.

CHAPTER

THIRTEEN

FASTER PYODIDE TESTING

We're going to do a lot of "sit in JS, execute Python." We want to speed it up. Let's change our test strategy to re-use a single Pyodide across many tests.

13.1 Clean up worker tests

We have a test that asserts the JS `initialize` function returns some flag. Actually, it's going to return the Pyodide instance.

Let's re-organize our worker tests:

```
import {beforeEach, expect, test} from "vitest";
import {initialize} from "../src/pyodide_components/worker.js";

// Make an interpreter and capture its startup state
const thisPyodide = await initialize();
const initialPyodideState = thisPyodide.pyodide_py._state.save_state();

beforeEach(async () => {
    // On each test, reset to an "empty" interpreter
    thisPyodide.pyodide_py._state.restore_state(initialPyodideState);
});

test("Load and initialize Pyodide", () => {
    expect(typeof thisPyodide.runPythonAsync).to.equal("function");
});

test("Confirm valid and running Pyodide", async () => {
    const result = await thisPyodide.runPythonAsync("1+1");
    expect(result).to.equal(2);
});
```

Lots of changes here. First, the two lines about `thisPyodide`. We make a Pyodide instance a module scope, then immediately capture its default startup state.

In `beforeEach`, we speed up our test runs. Instead of making a new interpreter all the time, we re-use the existing one. But we reset it to its initial state.

We then split our tests into two parts: did we actually get a Pyodide, and does it run Python code?

13.2 Worker returns a `pyodide`

Over in our `initialize` function, for now, just return Pyodide:

```
import {loadPyodide} from "./pyodide/pyodide.mjs";

export async function initialize() {
    return await loadPyodide();
}
```

13.3 Wrapup

We’re already in a neat spot. We can sit in a test, in a debugger, in our smart editor – and evaluate Python from JS.

CHAPTER

FOURTEEN

SIMPLE COMPONENT REGISTRY

The Python side needs to tell the browser side about registered custom elements, aka components. In this step, a Python function will return some data to the JS worker script.

14.1 Components in Python

We want coding in Python, then running in browser. But what is the unit of work that ties the two sides together? And what is the unit of sharing?

Components.

Stated differently, custom elements. With this, a web page can have HTML such as `<todo-list>My Items</todo-list>` that comes from Python code.

:::{note} Why not Web Components? Web components are a superset of custom elements, adding things such as a Shadow DOM. But this brings in sharp edges when used in practice. It isn't yet clear the gain is worth the pain. :::

We'll talk more about components later.

14.2 Python Registry

Let's write our registry.

In `tests/test_init.py`, add a failing test to get the registry. This test will have an import which fails While we're at it, we'll delete `test_hello` as we no longer need a placeholder.

```python
from pyodide_components import get_registry


def test_get_registry():
    assert get_registry() == [1, 2, 3]
```

Our tests fail, as expected. Let's now put in an implementation in `__init__.py`:

```python
def get_registry():
    return [1, 2, 3]
```

We'll similarly delete the PYODIDE_COMPONENTS flag, as it isn't needed to demonstrate testing.

:::{note} Why in `__init__.py`? It would be nicer to put this in, say, `loader.py`. But Pyodide doesn't yet have a good story for packages. You have to fetch every file yourself. Thus, we'll put all the Python in one file for now. :::

Tests now pass, let's hook this up to the JS side and do a test.

14.3 Fetch `src/pyodide_components/__init__.py`

When working with Pyodide – Python in the browser – you first have to fetch the Python code and put it in the virtual filesystem. You can then import it. Later, we'll have a wheel in PyPI and Pyodide can handle the installation.

The `initialize` function in `worker.js` gets some lines added to it. We're using `fetch` to make the HTTP request:

```
let loaderContent;
const response = await fetch("./__init__.py");
if (response.ok) {
    loaderContent = await response.text();
} else {
    console.log("Failed to fetch loader.py");
}
```

And this. . . fails badly:

```
TypeError: Failed to parse URL from ./__init__.py
    at Object.fetch (node:internal/deps/undici/undici:11118:11)
```

If you're using a NodeJS before 18.3.0 (when native fetch landed), you might get a different error.

What's going on? If you think about it, this code makes no sense from a NodeJS test. There isn't an HTTP server (and we really don't want one for testing.)

We need a way to intercept `fetch` and mock its responses. That's the topic for the next section.

CHAPTER

FIFTEEN

MOCKING FETCH

"Joyful" Pyodide dev means sitting in NodeJS tooling. But we're writing a browser app, which is going to issue HTTP requests using `fetch`. How do we replace the server? We'll start the process of using mocks.

15.1 Vitest config file

Our path to mocking begins with a small Vitest configuration file at `vitest.config.js` in the top-level folder:

```
import { defineConfig } from "vitest/config";

export default defineConfig({
  test: {
    setupFiles: ["./test/setup.js"],
  },
});
```

It's one job: point at a "setup" file which will mock our `fetch` request.

:::{note} Vitest, mocking, and MSW The Vitest mocking docs recommend using Mock Service Worker. Alas, that begins a rabbit hole. MSW doesn't support NodeJS native fetch). It instead relies on `node-fetch`, which requires absolute URLs. :::

15.2 Vitest setup test

We're about to write some JavaScript to mock `fetch` requests and responses. Let's do that by. . . writing a test!

Put this in `tests/setup.test.js`:

```
import {expect, test} from "vitest";
import {mockFetch} from "./setup.js";

test("File mapping works", async () => {
    const response = await mockFetch("./__init__.py");
    expect(response.ok).to.be.true;
    const responseText = await response.text();
    expect(responseText).to.contain("from pathlib");
});
```

The test fails, as we haven't started `tests/setup.js` yet.

15.3 Vitest setup file

Let's now write our Vitest setup file at `tests/setup.js`:

```
import {vi} from "vitest";
import {readFileSync} from "node:fs";

const INIT_PATH = "src/pyodide_components/__init__.py";
const INIT_CONTENT = readFileSync(INIT_PATH, "utf-8");

const FILES = {
    "./__init__.py": INIT_CONTENT,
};

export async function mockFetch(url) {
    if (url.includes("pyodide")) {
        return {
            ok: true,
            status: 200,
        };
    }
    const fileText = FILES[url];
    return {
        ok: true,
        status: 200,
        text: async () => fileText
    };
}

vi.stubGlobal("fetch", mockFetch);
```

This setup file is run whenever a test sessions starts up. It's useful for global initialization. Several notable points:

- `vi` installs `mockFetch` in the place of `fetch`
- `mockFetch` returns the file content from certain matched URLs

15.4 Implementation in `worker.js`

With this code in `worker.js`:

```
export async function initialize() {
    let loaderContent;
    const pyodide = await loadPyodide();
    const response = await fetch("./__init__.py");
    if (response.ok) {
        loaderContent = await response.text();
    } else {
        console.log("Failed to fetch loader.py");
    }
    return pyodide;
}
```

…our test now passes. You can see the `await fetch("./__init__.py")` line. When this is run under NodeJS and Vitest, the mock takes over. No HTTP request is sent. Instead, the contents of `src/pyodide_components/__init__.py` are read from disk and returned to the `fetch`.

15.5 Wrapup

We covered a lot of NodeJS weirdness in this step:

- `fetch` and versions of NodeJS
- Vitest configuration files
- Vitest setup files
- Installing mocks (and writing tests for them)

Truth be told, it's about the same as getting up-to-speed with `pytest` and network mocking. Ultimately, it's worth the investment.

CHAPTER

SIXTEEN

WORKER SCRIPT

Our `main.js` script runs in the main UI thread and has access to the DOM. We'll do the heavy lifting in a background "worker". But this will change the environment we have for testing.

In this step, we'll enable Vitest-based testing of "module workers".

16.1 Why and what of web workers

The Pyodide docs give a good rundown on the benefits of a web worker. Pyodide apps are a good candidate for this, to move heavy computation out of the main thread. As it turns out, the downside – communication has to be through JSON messages instead of object calls – can be a benefit for our design.

Web workers also introduce a top-level `self` object that acts somewhat like a `this`. It also can be very helpful. But it also comes with a downside.

16.2 `self` and NodeJS

This `self` value is part of browser JavaScript, not NodeJS. We will need a fake DOM, available in NodeJS, that gives us `self`, `addEventListener`, and other browser-centric APIs available in a worker.

We will use `happy-dom` as this NodeJS-based fake DOM. Install it:

```
$ npm install -D happy-dom
```

Then configure it according to the Vitest docs. We'll use it for all of our Vitest tests by adding `environment` to `vitest.config.js`:

```
test: {
    setupFiles: ["./tests/setup.js"],
    environment: "happy-dom"
},
```

16.3 The split-up

We want to handle specifically, we want to handle messages from the `main.mjs` module. We want those message handler functions to be easy to test. As such, we will have a `dispatcher` function which routes message "types" to callable functions. We then have an event listener to handle `postMessage` from the main thread, unpack the message data, and call the dispatcher.

This isolation lets us easily test without having to simulate the `postMessage` paradigm.

16.4 Message structure

The main module will send "messages" to the worker module, which will also send messages back. These messages need to be simple objects that can, essentially, be encoded as JSON.

For now, we'll just assume a "message" is an object with `messageType` and `messageValue`.

16.5 Worker dispatcher for unknown messages

We're going to write a function `dispatcher` which handles messages from the main thread. If it receives a message it doesn't know about, it will throw an exception.

We'll implement that part first. Starting with a test in `worker.test.js`:

```
import {initialize, dispatcher} from "../src/pyodide_components/worker.js";

  it("rejects an invalid messageType", async () => {
    const msg = { messageType: "xxx" };
    const error = await dispatcher(msg).catch((error) => error);
    expect(error).to.equal(`No message handler for "xxx"`);
  });
```

If your debug session with the Vitest watcher is still running, you'll see this fails. As expected: we haven't created `dispatcher` yet.

Let's create `src/pyodide_components/dispatcher.js` It's just a starting point:

export async function dispatcher({messageType, messageValue}) {

```
    throw `No message handler for "${messageType}"`;
}
```

16.6 Worker dispatcher for `initialize`

When the main module wakes up, it will create the worker and tell it to initialize Pyodide. It will do so via a message `{messageType: "initialize"}` and no `messageValue`. This is async, so it will expect to be sent a message `{messageType: "initialized"}`.

Let's write a test for this:

```
test("processes an initialize message", async () => {
    const msg = {messageType: "initialize"};
    const result = await dispatcher(msg);
    expect(result.messageType).to.equal("initialized");
});
```

We can now extend the `dispatcher` implementation:

```
export async function dispatcher({messageType, messageValue}) {
    if (messageType === "initialize") {
        await initialize();
        return {
            messageType: "initialized",
            messageValue: "Pyodide app is initialized"
        };
    }
    throw `No message handler for "${messageType}"`;
}
```

16.7 Keeping the pyodide around

Web workers also introduce a top-level `self` object that acts somewhat like a `this`. It also can be very helpful. But it also comes with a downside: this `self` value is part of browser JavaScript, not NodeJS.

We're going to need the `pyodide` instance in lots of places in worker. We could of course put it at global scope. Instead, we will use `self` as a place to store it.

Let's change the `Confirm valid and running Pyodide` test to assert that `self.pyodide` exists:

```
expect(self.pyodide).to.exist;
```

The test now fails. We then make a single-line change in `initialize`:

```
self.pyodide = await loadPyodide();
```

The test now passes. But we now have an issue: our watcher re-runs all the tests.

16.8 Preserve self.pyodide between test runs

Our problem: the `self.pyodide = await loadPyodide();` change above throws out the interpreter on every run. We'd like to keep the same `self.pyodide` between runs and just reset its state, as before.

First, let's change the `beforeAll` to get the worker's `self` to use a Pyodide from the test scope:

```
beforeEach(async () => {
    // On each test, reset to an "empty" interpreter
    thisPyodide.pyodide_py._state.restore_state(initialPyodideState);
    self.pyodide = thisPyodide;
});
```

Then, initialize should change to only assign a `self.pyodide` if it isn't present:

```
if (!self.pyodide) {
    self.pyodide = await loadPyodide();
}
```

Our tests are now fast again.

CHAPTER

SEVENTEEN

WORKER MESSAGING

Our worker has a dispatcher, but it isn't yet receiving or sending messages with the main module. Let's set up the `postMessage` machinery used in workers.

17.1 Receive messages

As explained in the MDN page for web workers, you handle incoming messages by defining an `onmessage` function. This can be at module scope or on the `self` variable, which is what we'll do.

We'll start of course with a test. We'll start small:

```
test("handles incoming messages with onmessage", async () => {
    expect(self.onmessage).to.exist;
});
```

This fails of course, so add the following to `worker.js`:

```
self.onmessage = async (e) => {
    // Unpack the message structure early to get early failure.
    const {messageType, messageValue} = e.data;
    const responseMessage = await dispatcher({messageType, messageValue});
    if (responseMessage) {
        self.postMessage(responseMessage);
    }
};
```

This code acts as a mediator between worker messaging and the dispatcher:

- Unpack the "protocol" of `messageType` and `messageValue`
- Call the dispatcher
- If the dispatched function returns something, post it back to the main module

This last point is a convenience. Our dispatched functions can always call `self.postMessage` themselves. But that's binding the unique mechanics of web workers into our logic.

We now have something to confront. We need to test calling `self.onmessage` and we want to see if `self.postMessage` was called. Even better, called with what we expect.

Mocking to the rescue.

17.2 Mocking `self.postMessage`

While `self` exists in `happy-dom`, `self.postMessage` doesn't. We need to do two things in `tests/setup.js`:

- Create a pretend aka "mock" implementation for `postMessage`
- Register it on global

Add the following to `tests/setup.js`:

```
if (!globalThis["worker"]) {
    self.postMessage = vi.fn();
}
```

This allows our code to pretend to call `self.postMessage`. Of course, nothing happens, but we don't care. Our `worker.test.js` code is in isolation, only interested in the worker, not the communication with main.

17.3 Testing `self.onmessage`

We now have the pieces in place. We can test the `self.onmessage` function which receives messages from main:

```
test("handles incoming messages with onmessage", async () => {
    expect(self.onmessage).to.exist;
    expect(self.postMessage).to.exist;

    // Make a fake worker message
    const event = new MessageEvent("message");
    event.data = {messageType: "initialize", messageValue: null};

    // Spy on self.postMessage, then call our handler
    const spy = vi.spyOn(self, "postMessage");
    await self.onmessage(event);

    expect(spy).toHaveBeenCalledWith({messageType: "initialized"});
});
```

- The first two `expect` just make sure our `self` has what we expect
- We then simulate the event object from the main<->worker communication
- Create a "spy" to watch the calling of `self.postMessage`
- Call `self.onmessage` and see if the resulting message back, matched what we thought

17.4 Layout of our worker

So what does this `worker.js` implementation look like? It's a 3-layer cake:

- The event listener which mediates with the main thread via messages
- The dispatcher, which finds the right message handler, calls it, and returns the message
- Handler functions which don't really know the outside world beyond `self`

CHAPTER

EIGHTEEN

WORKER TO MAIN

We’ve been in isolation. Let’s get back into the browser by hooking the worker module up to the main module.

18.1 What? Why?

Here are the layers of our cake:

- A web page includes the main module `main.js`
- Main makes a worker module that it sends/receives messages with
- The worker makes a Pyodide that it sends/receives calls with

In this step, we’ll do the second part. Our main module will:

- Initialize a `Worker`
- Send the worker a message, telling it to initialize Pyodide
- Receive a message from the worker, saying Pyodide is initialized
- Update the `document` with info from that message

We want to stay in the “joyful” mode by using tests. It will prove a little more complicated, as `happy-dom` doesn’t really support main<->worker. So we’ll still need to finish by confirming “end to end” in a browser. We’ll automate that in the next step with Playwright.

18.2 Cleanup

We’ll start by emptying `main.js` to delete the “tracer” constant we did earlier. Equally, we’ll edit `main.test.js` to delete that test.

18.3 Make an uninitialized `Worker`

Let’s create the worker. We’ll start with a test in `main.test.js`:

```
test("Worker initialization", () => {
    expect(Worker).to.exist;
});
```

And immediately, a problem. `Worker` is a global in browsers, but doesn’t exist in `happy-dom`. Thus, running this test produces:

```
ReferenceError: Worker is not defined
```

This is tricky as this happens as soon as we do an import. How can we inject a global before the import happens?

Back to `tests/setup.js` and our Vitest setup. Let's add a `MockWorker` and register it with Vitest's `vi.stubGlobal`:

```
const MockWorker = vi.fn(() => ({
    postMessage: vi.fn(),
}));

if (!globalThis["worker"]) {
    vi.stubGlobal("Worker", MockWorker);
    self.postMessage = vi.fn();
}
```

Back to `main.test.js`:

```
import {expect, test} from "vitest";
import {worker} from "../src/pyodide_components/main.js";


test("Worker initialization", () => {
    expect(Worker).to.exist;
    expect(worker.postMessage).to.exist;
});
```

Now an implementation in `main.js`.

```
export const worker = new Worker("worker.js", {type: "module"});
```

We created a `Worker` instance using the module worker option. Our test passes. Let's now implement an initializer.

18.4 Initialize worker

When the main module "wakes up" and makes the worker, it needs to tell the worker to "initialize". It appears to be simple: just add a `postMessage` call:

```
worker.postMessage({messageType: "initialize"});
```

Then, go to `main.test.js` and – like with `worker.test.js` – install a spy on `worker.postMessage`.

However, this reveals a flaw. `worker.postMessage` is at module scope and executes *at import*. It's already run before we get into a test and try to install a spy.

We need to refactor our code so that initialization doesn't happen at import time. As a note, this can have benefits to the application. Conceivably, the main UI module could "restart" the worker and thus Pyodide.

18.5 Delayed initialization

Let's change our tests to first assert that, at import time, `worker` exists but is uninitialized. Another test will call `initialize` and cause it to be assigned a `Worker` instance:

```
import {expect, test} from "vitest";
import {worker, initialize} from "../src/pyodide_components/main.js";


test("has no initialized worker", () => {
    expect(Worker).to.exist;
    expect(worker).not.to.exist;
});

test("initializes workers", () => {
    initialize();
    expect(worker).to.exist;
});
```

You might already spot the issue: how will the web page instrument the calling of `initialize`? We'll tackle that below.

18.6 Dispatcher table

The main module will send messages to the worker. But it will also receive messages. We'll have the equivalent of a dispatcher, using a little lookup table to find small message handler functions. These handlers can then be tested in isolation.

Let's write some failing tests:

```
import {worker, initialize, messageHandlers} from "../src/pyodide_components/main.js";


test("has handler lookup table", () => {
    expect(messageHandlers.initialized).to.exist;
});
```

Now in `main.js`, let's write the little dispatch table with an empty `finishedInitialized` handler:

```
export function finishedInitialize(messageValue) {

}

export const messageHandlers = {
    "initialized": finishedInitialize,
};
```

Our tests pass. Let's now move on to the dispatcher. We'll start with the easy part – handling invalid messages – which gets our first chunch in place.

18.7 Deal with invalid worker messages

This part is a bit complicated, as we'll ultimately have to install the `postMessage` mock again. We'll get the basics in place by focusing first on dealing with unknown worker messages.

First, a test for an unknown message and a test for "initialized":

```
import {dispatcher} from "../src/pyodide_components/worker.js";

test("rejects an invalid messageType", async () => {
    const msg = {messageType: "xxx", messageValue: null};
    const errorMsg = `No message handler for "xxx"`;
    await expect(async () => await dispatcher(msg)).rejects.toThrowError(errorMsg);
});
```

Now the `dispatcher` implementation:

```
export function dispatcher({messageType, messageValue}) {
    if (!(messageType in messageHandlers)) {
        throw `No message handler for "${messageType}"`;
    }

    messageHandlers[messageType](messageValue);
}
```

This lets us write a test for a valid message:

```
test("dispatches to finishedInitialize", async () => {
    const spy = vi.spyOn(messageHandlers, "initialized");
    const msg = {messageType: "initialized", messageValue: "Pyodide app is initialized"};
    await dispatcher(msg);
    expect(spy).toHaveBeenCalledWith("Pyodide app is initialized");
});
```

In this test, we spy on the `initialized` handler. We then ensure the dispatcher called it with the correct arguments.

18.8 Handler for `initialized`

Our `finishedInitialize` function is unimplemented. We'd like it to grab a `<span id="status"></span>` node in the document. Then, replace `innerText` with the `messageValue`.

First, a test. Our Vitest environment uses `happy-dom` as a fake DOM document. We need to initialize it to the HTML we expect in our real document. We'll add a `beforeEach` that sets up the document, then a test to ensure the `<span>` is there:

```
import {beforeEach, expect, test, vi} from "vitest";

beforeEach(() => {
    document.body.innerHTML = `<span id="status"></span>`;
});

test("document has a status node", () => {
    const status = document.getElementById("status");
```

(continues on next page)

(continued from previous page)

```
    expect(status).to.exist;
    expect(status.innerText).to.equal("");
});
```

With that in place, we can write a failing test:

```
test("updates document with initialized messageValue", async () => {
    const status = document.getElementById("status");
    const messageValue = "Loading is groovy";
    await finishedInitialize(messageValue);
    expect(status.innerText).to.equal(messageValue);
});
```

The implementation is really simple:

```
export function finishedInitialize(messageValue) {
    const status = document.getElementById("status");
    status.innerText = messageValue;
}
```

The test and implementation were simple for an important reason. We've adopted a development style where we can write small handler functions. These functions can be exported individually, then imported in a test.

The work is moved elsewhere for:

- Registering a message handler
- unpacking the agreed-upon message structure
- Finding the right handler
- Calling it with the right argument

18.9 Initialize when in a browser

Before we can open this in a browser, we have to confront a decision made above. Our `initialize` function isn't called anywhere. We could put `initialize()` at module scope. But it would then be executed by the test at import time.

We need a way of knowing if we are running under a test, inside Happy DOM. Let's arrange to set the "user agent". First, a failing test:

```
test("has correct user agent", () => {
    expect(navigator.userAgent).to.equal("Happy DOM");
});
```

Now in `tests/setup.js`:

```
navigator.userAgent = "Happy DOM";
```

The test passes. We can now add this to the end of `main.js`:

```
if (navigator.userAgent !== "Happy DOM") {
    // We are running in a browser, not in a test, so initialize.
```

(continues on next page)

(continued from previous page)

```
    initialize();
}
```

With this in place, we can finish our `main.js` with a finished `initialize`. The arrow function for `worker.onmessage` unpacks the data from the message before sending to the dispatcher.

```
export function initialize() {
    worker = new Worker("worker.js", {type: "module"});
    worker.onmessage = ({data}) => dispatcher(data);
    worker.postMessage({messageType: "initialize"});
}
```

18.10 Back into the browser

Let's see if things are working ok in the browser. All we really need to add is some HTML for the status message:

```
<div>Status: <span id="status">Startup</span></div>
```

With this in place, if you open `index.html` directly in a browser via an HTTP server, it works. But:

- Only in non-Firefox
- Not when bundling with Vite

Why not in Firefox? A 7-year-old unimplemented feature. Firefox implements web workers, but not *module* workers…meaning, you can't do ESM export/import in web workers. The ticket has recent activity.

In theory, a bundler like Vite is the answer. But Pyodide has some trouble with bundlers.

For the purpose of this series, we'll keep going and just view in Chrome/Safari, with no bundling.

CHAPTER

NINETEEN

PLAYWRIGHT PYODIDE

Hook up a Playwright test to ensure Pyodide winds up talking to the browser.

19.1 What? Why?

In an *earlier segment* we did actual browser testing, using Playwright. It's even more important now. We don't have a way to do end-to-end (E2E) testing, to see if Pyodide operations actually update the document.

Equally, the E2E part – integrating it all together – is kind of fiddly. Especially the bundler.

In this step, we'll learn to write a test for something that happens "later".

19.2 Test the page

Let's extend our `index` test in `test_pages.py`:

```python
def test_index(fake_page: Page):
    """Use Playwright to do a test on Hello World."""
    # Use `PWDEBUG=1` to run "head-ful" in Playwright test app
    url = "http://fake/index.html"
    fake_page.goto(url)
    assert fake_page.title() == "Pyodide Components"

    # When the page loads, the span is empty, until
    # Pyodide kicks in.
    span = fake_page.locator("#status")
    assert span.text_content() == ""

    # Now wait for the span to get some content
    text = "Pyodide app is initialized"
    span = fake_page.wait_for_selector(f"#status:has-text('{text}')")
    assert span.text_content() == text
```

The `.wait_for_selector` is the key. The locator waits until it finds something matching the selector.

One more small change. In `index.html`, change the `<script>` near the bottom:

```html
<script defer type="module" src="./main.js"></script>
```

The `defer` lets us avoid the `DOMContentLoaded` dance.

19.3 Wrapup

There’s another Playwright test we need to write in the future. The Vitest bundler output in `dist` is…well, kind of fiddly. In fact, at the time of this writing, it doesn’t work correctly.

Later, we’ll have Playwright tests that talk to the bundled output.

CHAPTER

TWENTY

REFACTOR SPEEDUP

Rewrite the Pyodide test speedup code to clear the local directory.

20.1 Why? What?

We previously sped up the Pyodide tests by retaining a Pyodide instance across test runs. When combined with Vitest "watch" and HMR, tests become essentially instant. Even when running under debug mode.

But we now have a flaw. We still want isolation on the `pyodide_components` code, as part of testing. We don't want the "local" filesystem, where things are imported from, to still have a directory with `__init__.py` etc. in it.

We also have a flaw with "self". It isn't part of the Pyodide state. It's the worker, as part of Happy DOM. Since we assign to `self`, we need to clear those assignments from call to call.

20.2 Removing the directory in `beforeEach`

Let's address that first in `beforeEach`, where we are currently restoring the startup, empty state of Pyodide.

In the first attempt, through JS and the `pyodide.FS.rmdir` and friends, we ran into all kinds of timing issues. It appears there is something async going on deep down in `emscripten`. A switch to Python fixed it. Here's the new `beforeEach`:

```
beforeEach(async () => {
  // On each test, reset to an "empty" interpreter
  thisPyodide.pyodide_py._state.restore_state(initialPyodideState);
  self.pyodide = thisPyodide;

  // If the pyodide_components directory exists, let's delete it
  // and start over
  const pathInfo = self.pyodide.FS.analyzePath("pyodide_components");
  if (pathInfo.exists) {
    self.pyodide.runPython(
      "import shutil; shutil.rmtree('pyodide_components')"
    );
  }
  self.pyodide.runPython("import os; os.mkdir('pyodide_components')");
});
```

20.3 Resetting `self`

In `beforeEach`, we'll also clear `self.pyodide_components` and `self.registry`:

```
// The "self" needs resetting
if (self.pyodide_components) delete self.pyodide_components;
if (self.registry) delete self.registry;
```

20.4 Wrapup

With this in place, we're now ready to return to building up the registry.

CHAPTER

TWENTYONE

REGISTRY REVISITED

Get the Python code into the browser and tell Pyodide to load it.

21.1 Why? What?

In *simple registry* we started the process of talking to the Python side. We copied the registry Python code but didn't apply it in any way.

We need to write this string to the filesystem, then import it. We'll stash the registry on `self.registry`, as a JS object literal.

And of course, we'll start through the lens of tests, to keep things "joyful".

21.2 First, a test

Remember, when this starts up, there's an empty Pyodide. The main module will message the worker, telling it to initialize `pyodide_components`. Thus, we'll start with a test that proves there is no `self.pyodide_components` (for the module) nor `self.registry`.

In `worker.test.js`:

```
test("starts with an empty pyodide_components and registry", async () => {
  expect(self.pyodide_components).not.to.exist;
  expect(self.registry).not.to.exist;
});
```

Now a test which initializes the Pyodide, then confirms that they exist:

```
test("initializes non-empty pyodide_components and registry", async () => {
  await initialize();
  expect(self.pyodide_components).to.exist;
  expect(self.registry).to.exist;
});
```

This test directly calls `initialize()`, which means it needs to be imported. But it's a nicer, more normal style of coding. No message-sending.

21.3 Registry with a `my-counter` component

Our registry is currently very dumb. Let's change it to have a single, hard-wired "component" called `my-counter`. Hard-wired means, the component is – for now – defined directly in the loader code.

First a test, to see if `get_registry` returns us a JS object shaped the way we want:

```
test("has MyCounter in registry", async () => {
  await initialize();
  expect(self.registry.length).to.equal(1);
  const myCounter = self.registry[0];
  expect(myCounter.get("name")).to.equal("my-counter");
});
```

We'll change the `__init__` implementation to just return a stub:

```
def get_registry():
    return [dict(name="my-counter")]
```

21.4 Wrapup

We now have our test strategy in place, with a good sequence for initializing everything. The registry just gave its first glimpse of storing custom element – aka component – definitions.

In the next step, we jump right into that.

CHAPTER

TWENTYTWO

SIMPLE COMPONENTS

We want custom elements, defined in Python, which we can use in HTML. In this step we arrange a proper registry which can communicate back to JavaScript. We also see the trick to create JS classes after startup.

22.1 Why? How

Our ultimate goal is to have `<my-counter count="0"></my-counter>` in our user's HTML. We want the definition – even the existence – of `<my-counter>` to be in Python.

But custom elements must be a JS class, registered in the `customElements` DOM object. How will we avoid making our developers write JS? Here's how:

- The Python side "discovers" component definitions
- Then, introspects them to build a little registry
- The JS side grabs that registry from the Python side
- For each entry, a custom class is created *dynamically*, at run time

We'll go a little further than that in the next step. But that's the strategy for now.

22.2 Discovery

We already have a test which confirms `my-counter` is in the registry. Let's write an implementation: an actual component, plus the discovery process.

Here's a simple, dataclass-based component to add in `__init__.py`:

```python
from dataclasses import dataclass

@dataclass
class MyCounter:
    pass
```

We could write a test for it, but at this stage, there's no real logic. We can trust that Python's dataclass machinery is already tested.

Next, let's write a tiny function that registers a component. We can see into the future and know – we'd like a helper to automate getting from `MyCounter` to `my-counter`. First, a test in `test_init.py`:

```
def test_to_element_case():
    result = to_element_case("MyCounter")
    assert result == "my-counter"
```

Not only does this test fail, but all of `test_init.py` fails. We'll focus our efforts in this test file.

The implementation in `__init__.py` is simple:

```
import re

def to_element_case(camel_str):
    """Convert MyCounter class name to my-counter custom element name."""
    return re.sub("([A-Z0-9])", r"-\1", camel_str).lower().lstrip("-")
```

22.3 Registration

"Something" will tell the system to put a component in the registry. First, let's define "the registry" as a global `defined_elements` dictionary. In `__init__.py`:

```
defined_elements = {}
```

A test to confirm it exists and starts empty:

```
def test_initial_globals():
    assert defined_elements == {}
```

Now a function `register` which is passed a component:

```
def register(component):
    element_name = to_element_case(component.__name__)
    defined_elements[element_name] = component
```

Once the imports are added, the test runs, but fails. Our `get_registry` is still hardwired. Let's fix that next.

22.4 Getting the registry

We'll circle back and fix the broken test which presumed the registry contained `[1, 2, 3]`. We'll also write a test that checks `get_registry`:

```
def test_get_registry():
    assert get_registry() == []

def test_register_new_component():
    assert get_registry() == []
    register(MyCounter)
    registry = get_registry()
    assert registry == [dict(name="my-counter")]
```

With this failing test, let's fix `get_registry`:

```python
def get_registry():
    return [
        dict(
            name=component_name,
        )
        for component_name, component in defined_elements.items()
    ]
```

This function now acts as a mediator between the JS side and the Python side. It dumps the registry into a format best-used in JS.

22.5 Loading the "app"

Let's revisit the layers of the cake:

- `index.html` loads the `main.js` main module
- The main module makes a worker module
- Main tells worker to initialize a Pyodide instance
- Worker tells main it has initialized Pyodide

We'll later introduce the idea of the "app" that will be loaded into Pyodide. For now, it's just a bundled `MyCounter` component. Main will need to tell the worker to load components.

Let's start with a test in `test_init.py`:

```python
def test_initialize_app():
    assert get_registry() == []
```

Hmm, this ended quickly. We haven't registered anything yet – why is this test failing.

Because it still has the state from the previous registration. Remember, our `defined_elements` "database" is a global.

We'll need a test and implementation for resetting the registry:

```python
def test_reset_registry():
    """Clear previous test's registration."""
    reset_registry()
    assert get_registry() == []
```

Then, in `__init__.py`, the reset function *and* the `initialize_app`:

```python
def reset_registry():
    """Used by tests to put the globals back in original state."""
    defined_elements.clear()

def initialize_app():
    register(MyCounter)
```

Now we use the reset in our test, and with the proper imports, it passes:

```python
def test_initialize_app():
    reset_registry()
    assert get_registry() == []
```

(continues on next page)

(continued from previous page)

```
    initialize_app()
    assert get_registry() == [dict(name="my-counter")]
```

We don't want to have to do this reset dance all the time so we'll write a pytest fixture for later use:

Let's write a pytest fixture:

```
import pytest

@pytest.fixture
def initialized_app():
    """Reset the registry and setup counter app."""
    reset_registry()
    initialize_app()
```

22.6 Worker initializes components

When the worker starts, there is no Pyodide. The main module sends a message saying "initialize Pyodide", which returns a message when it is done. In that return message, we want to then say "initialize the app", where "app" is a collection of Pyodide Components.

At them moment, `worker.test.js` fails. It's expecting `my-counter` to already be in the registry. But we just made it a manual, explicit step: load Pyodide, *then* load components.

First, a test in `worker.test.js` for the `loadApp` function itself:

```
test("has MyCounter in registry", async () => {
  await initialize();
  expect(self.registry.length).to.equal(0);
  await loadApp();
  expect(self.registry.length).to.equal(1);
  const myCounter = self.registry[0];
  expect(myCounter.get("name")).to.equal("my-counter");
});
```

Remember to import `loadApp`. Next, an implementation:

```
export async function loadApp() {
  self.pyodide_components.initialize_app();
  self.registry = self.pyodide_components.get_registry().toJs();
  return {messageType: "finished-loadapp", messageValue: self.registry};
}
```

With this, the test passes. One more step: we need a handler for the message dispatcher.

```
test("processes a load-app message", async () => {
  await initialize();
  const msg = { messageType: "load-app" };
  const result = await dispatcher(msg);
  expect(result.messageType).to.equal("finished-loadapp");
});
```

We need to make a change to `dispatcher` to handle a `load-app` message:

```
  if (messageType === "initialize") {
    await initialize();
    return {
      messageType: "initialized",
      messageValue: "Pyodide app is initialized",
    };
  } else if (messageType === "load-app") {
    return await loadApp();
  }
```

The test passes. Main is able to send the worker a `load-app` message and receive back an updated registry.

22.7 A test for custom elements

Let's now hook this up to the main module and allow `<my-counter>` to exist in HTML.

First, a failing test. We'll do so as part of `main.test.js`. Add to the `beforeEach` a usage:

```
beforeEach(() => {
  document.body.innerHTML = `<span id="status"></span><my-counter id="mc1">Placeholder</
↪my-counter>`;
});
```

A test to see that this node exists, with `Placeholder` as the content:

```
test("has a placeholder my-counter", async () => {
  const status = document.getElementById("mc1");
  expect(status.innerText).to.equal("Placeholder");
});
```

That's a good start. Let's start writing the part that makes face custom element classes on the fly, then hooks them into the registry messaging.

22.8 Fake custom element classes

Ok, here's the fun part: dynamic custom elements! We will have a `makeCustomElement` function that acts as a factory. You call it with the name you want – such as `my-counter` – and it returns a *class*. Our message handler will then register that class as a custom element.

First, a test:

```
test("construct a custom element", () => {
  const factory = makeCustomElement("my-counter");
  expect(factory).is.a("function");
  const element = new factory();
  expect(element.name).to.equal("my-counter");
});
```

Now, an implementation:

```
export function makeCustomElement(name) {
  return class extends HTMLElement {
    constructor() {
      super();
      this.name = name;
    }

    connectedCallback() {
      this.innerHTML = `<div>Element: <em>${this.name}</em></div>`;
    }
  };
}
```

We're close! Now we need to wire this up to the `customElement.define` function.

22.9 Put elements in the custom element registry

The main module will receive a `finished-loadapp` message when the registry is updated. Let's implement that, but first, with a test:

```
test("initialize the registry", () => {
  expect(window.customElements.get("my-counter")).not.to.exist;
  const thisEntry = new Map();
  thisEntry.set("name", "my-counter");
  finishedLoadApp([thisEntry]);
  expect(window.customElements.get("my-counter")).to.exist;
});
```

And now, with an implementation of `finishedLoadApp`:

```
export function finishedLoadApp(registryEntries) {
    // When an app loads components, the worker gives us an updated registry.
    registryEntries.forEach((entry) => {
        const name = entry.get("name");
        customElements.define(name, makeCustomElement(name));
    });
}
```

And the test now passes. We have defined a custom element in the custom element registry.

22.10 Get the custom element innerHTML

Our tests have a `document` with HTML setup in `beforeEach`. Is the placeholder text replaced with the `connectedCallback` text? Let's write a test:

```
test("find the custom element innerHTML", () => {
  expect(window.customElements.get("my-counter")).not.to.exist;
});
```

Hmm, failed quickly. We lost test isolation again, because `window.customElements` – which is an instance of `CustomElementRegistry` – is already popuplated. Let's fix that first by resetting the Happy DOM `window` in `beforeEach`:

```
beforeEach(() => {
  window = new Window();
  document.body.innerHTML = `<span id="status"></span><my-counter id="mc1">Placeholder</
↪my-counter>`;
});
```

That test now passes. Now finish the test to see if we can trigger `connectedCallback`:

```
test("find the custom element innerHTML", () => {
  expect(window.customElements.get("my-counter")).not.to.exist;
  const thisEntry = new Map();
  thisEntry.set("name", "my-counter");
  finishedLoadApp([thisEntry]);
  document.body.innerHTML = `<my-counter id="mc1">Placeholder</my-counter>`;
  const mc1 = document.getElementById("mc1");
  expect(mc1.innerHTML).to.equal("<div>Element: <em>my-counter</em></div>");
});
```

It was a little finicky, but…we were able to do custom elements in Happy DOM, without a Chromium browser. To wrap up, let's register the message handler for the worker's message to the main module:

```
export const messageHandlers = {
  initialized: finishedInitialize,
  "finished-loadapp": finishedLoadApp,
};
```

Then, in `finishedInitialize`, the main needs to tell the worker to load the app:

```
export function finishedInitialize(messageValue) {
  const status = document.getElementById("status");
  status.innerText = messageValue;
  worker.postMessage({
    messageType: "load-app",
  });
}
```

22.11 Wire into index.html

Let's see if we can get a working web page, in a browser. We'll add this in the body:

```
    <div>
      <my-counter></my-counter>
    </div>
```

Now let's wrap up with a Playwright E2E test. We'll add to the test in `test_pages.py`:

```
    # Did the custom element render into the innerHTML?
    my_counter = fake_page.wait_for_selector("my-counter em")
    assert my_counter.text_content() == "my-counter"
```

CHAPTER

TWENTYTHREE

LOADABLE APPS

Our “counter” app is currently bundled into “the system”. Let’s move it to `counter.py` and then teach “the system” to load “apps”.

23.1 Why? How?

In a theoretical finished system, “pyodide_components” would be in a package distributed on PyPI. People would then need a way to point it at their app.

Of course, we’re just writing a series of articles, not planning any shipping software. Still, it’s a useful aid, to help reason about the boundaries between things.

We’ll construct it like this:

- Main messages the worker, telling it to load a named app
- Worker fetches that app’s Python file, writes it to disk, imports it
- Worker then tells “the system” to initialize the app with that app module
- Worker then messages main with the updated registry

With this, we’ll late be able to see a good DX for writing pluggable apps.

23.2 Refactor into own module

We’ll start by extracting `MyCounter` into `counter.py`. We start here because it will help find the flaws in our hardwired approach. Our tests will break, and we’ll have to fix them to remove the assumption.

```
from dataclasses import dataclass


@dataclass
class MyCounter:
    pass
```

Along the way, `initialize_app` needs to comment out the line that registers `MyCounter`.

Our `test_init.py` file has an import of `MyCounter`. We comment out the `test_initialize_app` test. We’re left with just the failed test for the initialized registry. Good start.

23.3 Teach `setup.js` to fetch `counter.py`

Remember when we patched `fetch` to return content from disk, rather than issue an HTTP request? This was in `tests/setup.js`. We're still doing it in a dumb way, so continue on:

```
const COUNTER_PATH = "src/pyodide_components/counter.py";
const COUNTER_CONTENT = readFileSync(COUNTER_PATH, "utf-8");

const FILES = {
    "./__init__.py": INIT_CONTENT,
    "./counter.py": COUNTER_CONTENT
};
```

23.4 Fix the `loadApp`

We're going to teach `loadApp` in the worker to:

- Be passed an app name
- Fetch the Python file
- Write to local disk
- Import

We'll start in `worker.test.js`. First, in `has MyCounter in registry`, change the `loadApp` call:

```
  await loadApp({appName: "counter"});
```

Our `loadApp` function now changes. Again, this can all be done in smarter ways:

```
export async function loadApp({ appName }) {
  let appContent;
  const response = await fetch(`./${appName}.py`);
  if (response.ok) {
    appContent = await response.text();
  } else {
    console.log(`Failed to fetch ${appName}`);
  }
  self.pyodide.FS.writeFile(`${appName}.py`, appContent, { encoding: "utf8" });

  // Python timestamp thing with MEMFS
  // https://github.com/pyodide/pyodide/issues/737
  self.pyodide.runPython("import importlib; importlib.invalidate_caches()");
  const appModule = self.pyodide.pyimport(appName);

  // Now register the app and update the local registry
  self.pyodide_components.initialize_app(appModule);
  self.registry = self.pyodide_components.get_registry().toJs();

  return { messageType: "finished-loadapp", messageValue: self.registry };
}
```

You might notice self.pyodide_components.initialize_app(appModule). Previously, initialize_app was passed an argument. So we head over to __init__.py to change the protocol.

23.5 Initialize apps

We're about to do what we intended to do originally – initialize an *app*.

Head to test_init.py and let's make it look the way the API should work. First, let's change the fixture to initialize the registry with the counter app:

```python
@pytest.fixture
def initialized_app():
    """Reset the registry and setup counter app."""
    reset_registry()
    initialize_app(counter)
```

This requires an import of counter. We now do an implementation of initialize_app:

```python

def initialize_app(app_module):
    """Scan for known components and register them."""
    setup_function = getattr(app_module, "setup_pyodide")
    setup_function(register)
```

We're doing something different here. We expect the passed-in module to have a setup_pyodide function. "The system" grabs that function and calls it, passing in its register function.

This means a trip to counter.py to add in that function:

```python

def setup_pyodide(register):
    """Run the register function to set up component in this app"""
    register(MyCounter)
```

With this in place, test_init.py now passes all tests.

23.6 Finish wiring up worker

Some minor changes now needed. dispatcher needs to send the messageValue to loadApp. Our tests then need to send a full message to dispatcher:

```javascript
const msg = { messageType: "load-app", messageValue: { appName: "counter" } };
```

23.7 Tell `main` to load the `counter` app

Our `main.js` is now ready to kick things off. Change the `finishedInitialize` handler to post the correct message:

```
worker.postMessage({
  messageType: "load-app",
  messageValue: { appName: "counter" },
});
```

All of our JS and Python tests pass.

23.8 Wrapup

We still aren't in a complete "loadable app" setup. You still have to edit `main.js` to decide what app to load. We could, for example, move it to a well-known place, such as a data attribute on `body`.

But we're not writing a ready-to-go app, just a series of docs.

CHAPTER

TWENTYFOUR

COMPONENT INSTANCES

Have Python track instances of components, then dispatch events from the JavaScript side.

24.1 Why track instances?

Answer: state!

Let's say we have a counter component. It needs to know the current count: to display, and to increment. Each usage of `<my-counter>` is a different, well, counter and thus a different instance.

We want the state to be in Python, not in JavaScript. The JavaScript side sends messages to the Python side, which will return HTML to update the usage. We'll use a shared UID to connect the JavaScript and Python instances.

:::{note} No attributes or events We're still at this point not worried about attributes (aka props) nor events. :::

24.2 Python test and code

We'll start by adding some state to `MyCounter`, plus two methods that we'll use. Test first, of course:

```python
import pytest

from pyodide_components.counter import MyCounter


@pytest.fixture
def this_component():
    return MyCounter(uid="n123")


def test_increment(this_component: MyCounter):
    assert this_component.count == 0
    this_component.increment()
    assert this_component.count == 1


def test_onclick(this_component: MyCounter):
    assert this_component.count == 0
    this_component.onclick({})
    assert this_component.count == 1
```

(continues on next page)

(continued from previous page)

```python
def test_render(this_component: MyCounter):
    html = this_component.render()
    assert "<span>0" in html
```

Now the implementation. Our instances will be assigned a `uid` determined on the JS side.

```python
@dataclass
class MyCounter:
    uid: str
    count: int = 0

    def increment(self):
        self.count += 1

    def onclick(self, event):
        self.increment()

    def render(self):
        # language=html
        return f"<p><strong>Count</strong>: <span>{self.count}</span></p>"
```

We want a "database" to track these instances, by UID. Back to `test_init.py`. We'll ensure that the global for the db starts empty, which also means an import:

```python
def test_initial_globals():
    assert defined_elements == {}
    assert db == {}
```

Next, add a test for the `make_element` machinery we are about to add:

```python
def test_make_and_update_element(initialized_app):
    html = make_element("n123", "my-counter")
    assert "<span>0" in html
    my_counter = db["n123"]
    my_counter.increment()
    assert "<span>1" in my_counter.render()


def test_make_and_update_element_with_prop(initialized_app):
    """The node had an HTML attribute."""
    html = make_element("n123", "my-counter")
    assert "<span>0" in html
    my_counter = db["n123"]
    my_counter.increment()
    assert "<span>1" in my_counter.render()
```

We'll be adding a `make_element` message, sent from main to the worker. While we're at it, add db to our `reset_registry` function:

```python
db = {}
```

(continues on next page)

(continued from previous page)

```python
def reset_registry():
    """Used by tests to put the globals back in original state."""
    defined_elements.clear()
    db.clear()

def make_element(uid, name):
    """Receive message from JS and make a node"""
    factory = defined_elements[name]
    instance = factory(uid)
    db[uid] = instance
    return instance.render()
```

With that, our two new tests pass. That means we have a way for the worker to tell our Python system to make new component instances *and* render them.

Over to the worker.

24.3 Tell the worker to tell Pyodide

How does a component instance get created? What is it that calls `make_element`?

We need to make a Python instance *during custom element insertion* into DOM. That means, in the connectedCallback, whic means the message actually originates in the main module. It will then `postMessage` to the worker.

24.4 Refactor worker message handling

The main module has a nice "handlers" table for message dispatch. The worker message dispatching works on an `if` basis. This was different as the hope was to just make the message name match the Python function name.

But this doesn't quite work. We need a mediator, to unpack the `messageValue` and prepare arguments for the Python function. Let's convert `worker.mjs` to use a `handler` table:

```javascript
const messageHandlers = {
  initialize: initialize,
  "load-app": loadApp,
};

export async function dispatcher({ messageType, messageValue }) {
  if (!(messageType in messageHandlers)) {
    throw `No message handler for "${messageType}"`;
  }
  const handler = messageHandlers[messageType];
  const result = await handler(messageValue);
  if (result) {
    return {
      messageType: result.messageType,
      messageValue: result.messageValue,
    };
  }
}
```

We then change the return value of `initialized`:

```
  return {
    messageType: "initialized",
    messageValue: "Pyodide app is initialized",
  };
```

24.5 `make-element` message

We'll tackle the last part now. A test, to dispatch a `makeElement` message that calls that function.

```
test("makes a new element", async () => {
  await initialize();
  await loadApp({ appName: "counter" });
  makeElement({ uid: "n123", name: "my-counter" });

  const expected = {
    messageType: "render-node",
    messageValue: {
      uid: "n123",
      html: "<p><strong>Count</strong>: <span>0</span></p>",
    },
  };
  expect(self.postMessage).toHaveBeenCalledWith(expected);
  const thisDb = self.pyodide_components.db.toJs();
  expect(thisDb.get("n123").uid).to.equal("n123");
});
```

Remember, our `setup.js` file helpfully puts a mock on the worker's `postMessage`. This lets us see what it was telling the main module. We see that our "component" rendered itself. We also poke into the Python side to see that a component *instance* was stored in the db.

Let's now implement `makeElement`:

```
export function makeElement({ uid, name }) {
  // Post a message to Pyodide telling it to make a node then render
  const html = self.pyodide_components.make_element(uid, name);
  renderNode(uid, html);
}
```

Add it to the handlers table:

```
const messageHandlers = {
  initialize: initialize,
  "make-element": makeElement,
  "load-app": loadApp,
};
```

However, the test fails. Our new `makeElement` function runs, but it calls a `renderNode` function. This doesn't exist – what's that?

24.6 Rendering the component output

Our components render HTML in *Python* and returns to the JS function that calls it. The JS side needs to take that and update the document. But it's in the worker, which has no access to the DOM. So the worker needs to message the main module: "Change this node to have this HTML."

There's not much to the implementation. It simply wraps up inputs and does `postMessage`:

```
function renderNode(uid, html) {
    self.postMessage({
        messageType: "render-node",
        messageValue: {uid, html},
    });
}
```

With this, the tests pass.

24.7 main.mjs

On to actually updating the document. We have a dynamically-generated anonymous class. Each instance of that needs to generate a `uid` *data* attribute and store it on `this` in the constructor. We don't want to use `id` as that should be left for the user.

Then, the `connectedCallback` method posts the `make-element` message to the worker. When the worker makes the element instance in Python, it then posts back the `render-node` message.

Here are the changes to the dynamic class:

```

```

Next, handle the `render-node` message from the worker:

```
import * as Idiomorph from "./static/idiomorph.js";

function renderNode({uid, html}) {
    const target = document.querySelector(`*[data-uid=${uid}]`);
    Idiomorph.morph(target, html, {morphStyle: "innerHTML"});
}

export const messageHandlers = {
  initialized: finishedInitialize,
  "finished-loadapp": finishedLoadApp,
  "render-node": renderNode,
};
```

What's the `Idiomorph` line? Idiomorph is a DOM-merging library. Let's copy the file from the repo and save it as `idiomorph.js`.

24.8 E2E test update

We wrap up by heading to the Playwright test in `test_pages.py` and updating our end-to-end (E2E) test:

```python
# Did the custom element render into the innerHTML?
my_counter = fake_page.wait_for_selector("my-counter span")
assert my_counter.text_content() == "0"
```

Good news, our rendering-from-Python made it back through the worker, into the main module and browser.

24.9 Wrapup

With that in place, we have component rendering in Python.

CHAPTER TWENTYFIVE

TODO

Below is a list of some to do items for future work and the next rewrite. Here's the list of sections that have already been written, but not put into this update:

events attributes detatch mocks apps mock_fetch_pyodide server prerender

25.1 Next

- Load the **init**
- Dispatcher with messages that go straight to Python
- A mock-able self.pyodide.*
- self.registry
- Rewrite
  - Setup Playwright tests from the beginning
  - Make sure Vite and Firefox work
  - Include the dist dir
  - Move the `fetch` part to much later
  - Emphasize coverage
  - Less Markdown, more comments
- For JS, unit test vs. integration tests…markers? Separate file names?
  - Unit tests all have a fake `self.pyodide_components`
  - This will speed them up and make them more reliable
- Replace the passed-in Python string with mocked requests
- Create mocks for executing Pyodide for faster JS tests
- Have All Tests flavor that excludes slower tests (like pytest.mark)
- minx runner
- App
- Injector